Integration of the Reconfigurable Self-Healing eDNA Architecture in an Embedded System

Michael Reibel Boesen[1], Didier Keymeulen[2], Jan Madsen[1], Thomas Lu[2], Tien-Hsin Chao[2]

[1] Technical University of Denmark
Richard Petersens Plads, Bygning 322
Kgs. Lyngby, Denmark 2800
+45 31596369
{mrb,jan@imm.dtu.dk}

[2] Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109
818-354-4280
{didier.keymeulen,thomas.t.lu,tien-hsin.chao@jpl.nasa.gov}

Abstract—In this work we describe the first real world case study for the self-healing eDNA (electronic DNA) architecture by implementing the control and data processing of a Fourier Transform Spectrometer (FTS) on an eDNA prototype. For this purpose the eDNA prototype has been ported from a Xilinx Virtex 5 FPGA to an embedded system consisting of a PowerPC and a Xilinx Virtex 5 FPGA. The FTS instrument features a novel liquid crystal waveguide, which consequently eliminates all moving parts from the instrument. The addition of the eDNA architecture to do the control and data processing has resulted in a highly fault-tolerant FTS instrument. The case study has shown that the early stage prototype of the autonomous self-healing eDNA architecture is expensive in terms of execution time.

TABLE OF CONTENTS

1. INTRODUCTION

In the age of ubiquitous computing, all parts of the industry is in need of highly robust hardware platforms. Embedded systems are given increasingly often life-saving, life-depending roles, such as autonomous subway systems, airplanes, cars, hospital equipment etc. An unprotected hardware fault in any of these will have dire consequences, consequently hardware faults in such systems are always protected by huge amounts of redundancy. But even the state-of-the-art hardware fault prevention technique – Triple Modular Redundancy (TMR) has its limits. A fault in the voter circuits or a permanent fault in one of the copies will eliminate the TMR's ability to identify the correct value, while a repair of the faulty module will allow it to reconstruct the TMR. The capability of a hardware platform to repair itself becomes particularly useful in space, where a repair mission will be either a great risk, impossible or very expensive, or all of the above.

In the last decade, several biologically inspired reconfigurable self-healing hardware platforms have been proposed [3,4,5,6]. All of these suffer from problematic scaling issues due in particular to a too low level of logical granularity. Consequently, (to the best of our knowledge) neither of these has ever been applied to a real world application.

Other approaches, such as roving STARS [7] uses a centralized approach, where a centralized processing unit is responsible for performing the fault tolerance mechanism. Clearly, approaches using a centralized unit have single-point-of-failure properties, which in a high reliability environment would be unacceptable.

The eDNA (electronic DNA) architecture [1,2] is designed for an ASIC implementation and will consequently be an entirely new type of fault-tolerant coarse grained FPGA due to the increased level of logical granularity, when compared to other approaches. The increased level of logical granularity makes the cost of the self-healing feature acceptable [2].

The case study application studied is a Fourier Transform Spectrometer (FTS) application. The prototype of the instrument is produced by NASA's Jet Propulsion Laboratory and Vescent Photonics Inc. [10,11]. The instrument itself takes a novel approach to mechanical

[1] 978-1-4244-7351-9/11/$26.00 ©2011 IEEE.
[2] IEEEAC paper#1453, Version 4, Updated 2010:10:26

robustness due to its novel liquid crystal waveguide, which eliminates the need for moving parts in the instrument. The moving parts in an FTS are considered the highest risk in sending an FTS instrument to space, due the extreme g-forces applied during launch. This makes this case study particularly interesting because the result of applying the eDNA architecture to such a system would be an FTS instrument which is not only robust to mechanical faults but also to permanent and transient hardware faults.

The next section outlines the basic structure and concept of the eDNA concept. After this, section 3 goes deeper into the concept and the architecture of the eDNA prototype. Section 4 describes in detail the FTS application. Section 5 describes the integration of the FTS application on the eDNA architecture as well as the integration of the eDNA architecture on the embedded system. Section 6 presents the performance analysis of the eDNA architecture resulting from running the application. Finally, section 7 presents our conclusion.

2. eDNA: Fundamental concepts

eDNA system is the name of the entire package described in this paper – consequently, we have two fundamental terms; the eDNA *architecture* and the eDNA *program*. Figure 1 shows an overview of the entire eDNA package.

The eDNA architecture consists of a distributed array of multiple homogenous processing units called *electronic cells* (eCells). The term *homogenous* expresses the fact that all eCells contains the same hardware. The job of all eCells combined, is to implement the eDNA program, which is specified by the programmer. The eDNA program is translated into a binary version of the program, which is then fed to all eCells which all store it in a RAM block. Each eCell implements a part of the eDNA program. The specific part, which an eCell implements is, called the *gene* of this particular eCell.

Each eCell contain a microprocessor and a 32 bit ALU which is configured by the microprocessor to perform a certain function described by the gene. The program run by the microprocessor is termed the *ribosomal DNA* (referring to the intracellular organelle in biological cells, responsible for synthesizing proteins and consequently functionality of the cells). The ribosomal DNA is a program written for the eCell microprocessor, which performs *self-organizing* and *self-healing* of the eDNA architecture. All eCells contain a copy of this program.

Observe that no centralized processing unit is present. The eCells cooperate to complete the self-organization and self-healing completely automous and without "outside" help.

The eCells communicate with each other through a Network-on-Chip (NoC) 2D-mesh-8 architecture, where each eCell communicates directly with at most 8 adjacent neighbors depending on position. The position of an eCell in the NoC is represented by an (X,Y)-coordinate set. The NoC completes package transfers between eCells using a fault-tolerant data-transfer protocol, which can route around dead links.Figure 1 shows an overview of the eDNA package.

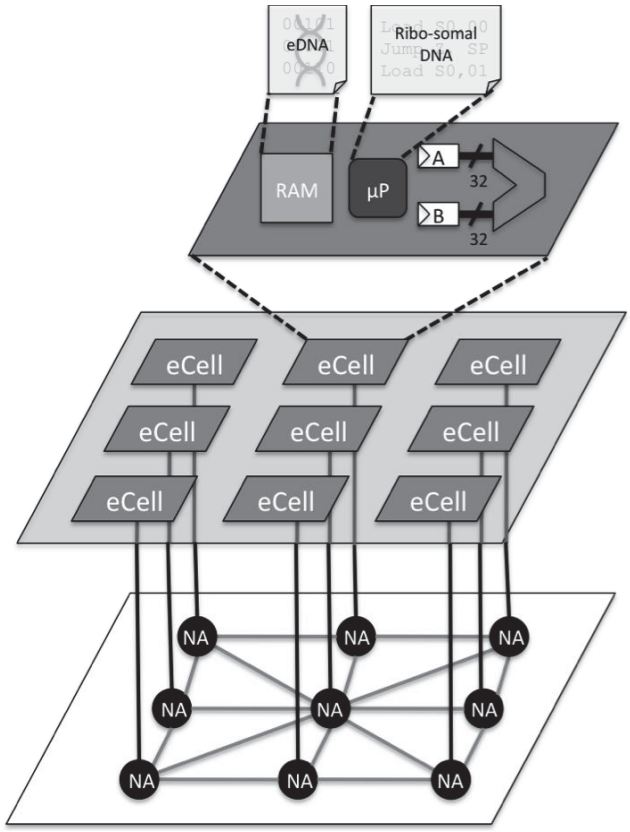

Figure 1 Overview of the eDNA architecture

3. eDNA architecture

The electronic DNA (eDNA) hardware architecture presents a solution to this problem. Inspired by biology, the eDNA architecture consists of an array of homogenous small CPUs known as electronic cells (eCells) [1,13]. In our prototype implementation [2] each eCell consists of a Xilinx PicoBlaze (PB) [8,9], which is an 8-bit VHDL synthesizable soft-core provided by Xilinx (see Figure 2). The PicoBlaze is only responsible for executing the self-organizing and self-healing algorithms (to be described later), which are located in the local RAM of the PicoBlaze. At runtime when no faults occur a 32-bit ALU is used for dataprocessing, where it performs an operation on the two 32-bit operands *A* and *B*. A detailed block diagram of an eCell can be seen in Figure 2.

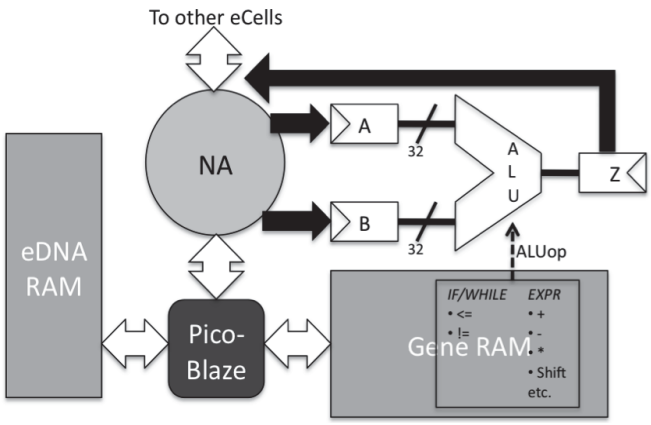

Figure 2 eCell Block Diagram

The eCells communicate with each other via a 2D-mesh type Network-on-Chip (NoC) infrastructure. This allows the eCells to establish dynamical paths with each other. The package transfer in the network is implemented with a fault-tolerant handshaking protocol, which is capable of routing around dead links. An overview of the different parts of the eDNA architecture can be seen in Figure 3.

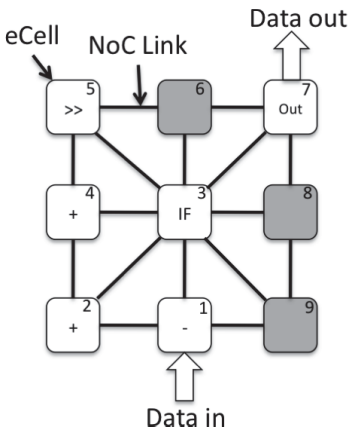

Figure 3 Schematic of 3x3 eDNA architecture

Self-organization: Programming eDNA

The eDNA architecture is programmed by a program (the eDNA program) written in the eDNA programming language. The eDNA programming language [1,2] is a basic programming language with the usual if-then-else and loop control structures, but also includes an explicit way for the programmer to indicate which program parts should be run in parallel.

During initialization each eCell receives a binary encoded version of the eDNA program and stores it in its local RAM block (the block RAM denoted *eDNA RAM* in Figure 2). In this way each eCell contains its copy of the eDNA program, this gives the eCells the powerful feature of self-awareness – i.e. each eCell know exactly what piece it plays in the complete applicational puzzle as well as what roles other eCells play. The binary encoded version of the eDNA program divides the eDNA program written in the eDNA language into tasks known as genes. Each eCell implements one gene (one task), where one gene is defined as one arithmetic operation of the eDNA program. An example of an eDNA program is seen in Figure 4.

```
if (A == B) then
   A = B + C
else
   B = B - C
endif;
C = B + 2
```

Figure 4 Example eDNA program

This eDNA program would require 4 eCells: One to implement `if (A == B) then`, one to implement `A = B + C`, one to implement `B = B - C`, and one to implement `C = B + 2`.

Self-organization: Gene Mapping and Placement

When all eCells have received the complete eDNA program the self-organizing begins. The self-organizing is where each eCell locates its gene and configures the 32-bit ALU accordingly, by moving the particular genes to a block RAM shown in Figure 2 as the *Gene RAM*. Each eCell locates its gene by using its unique identifier, which identifies its position in the array. This identifier is simply an integer from 0 to N-1 (where N is the number of eCells) (seen in Figure 3 as the integer in the corner of each eCell). The identifier is distributed together with the eDNA. This identifier is consequently responsible for mapping the functionality of the eDNA program onto the different eCells. The self-organizing algorithm running on the PicoBlaze of an eCell is implemented by counting the number of genes and comparing it to the identifier this particular eCell has.

Prior to this our eDNA software toolkit has analyzed the mapping of the application using a metaheuristic algorithm. The metaheuristic algorithm will return a potentially optimal solution to the mapping problem in the form of a mapping of genes to eCells, i.e., the sorted list of identifiers of eCells. The eCells then implement this mapping when self-organizing.

Self-healing

Self-healing is done by moving the gene of a faulty eCell to a spare eCell, i.e., an eCell which hasn't been assigned a gene yet. Note that no physical "gene moving" takes place as the spare eCell already have the complete eDNA. Only the mapping of that particular gene switches from the faulty eCell to the spare eCell.

A crucial part of the self-healing process is the fault-detection. However, in the current state of the prototype we haven't implemented the fault-detection algorithm yet, so in this study we rely on manual fault-injection. When a fault is injected the self-healing algorithm run by the PicoBlaze of an eCell, completes the following steps:

1. Restore the gene of the faulty eCell at a spare eCell
2. Restore the gene state at the spare eCell

Step 1 is the restoration of the functionality of the faulty eCell at a spare eCell. It is completed by simply remapping the unique identifier of the faulty eCell to a spare eCell and then broadcasting a message to all eCells requesting them to run the self-organizing algorithm again. This will cause the non-faulty eCells to speak with the spare eCell instead of the faulty eCell and will cause the spare eCell to realize that it is now an active eCell and participating in the running the application.

Step 2: The gene state is basically the two registers holding the operands of the eCells functionality, e.g., B and C of the `A = B + C` eCell in the example program from Figure 4. Recovering the gene state is inherent to our fault detection algorithm and since it is not implemented in the prototype yet, it is not the scope of the paper. In the present prototype we can only recover registers B and C if they are constants. Otherwise, we might "be lucky" that the application will reinitialize the non-constant with the right value before the eCell is executed again.

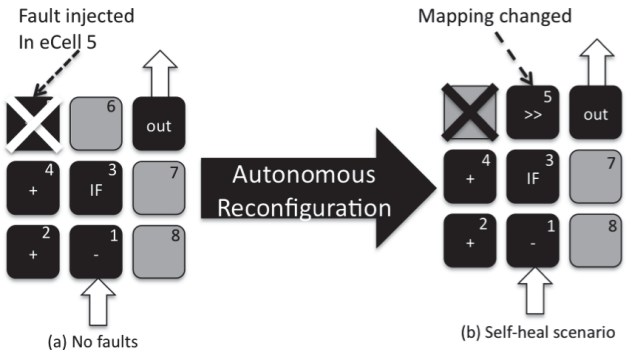

Figure 5 Self-healing example where 3 eCells are faulty and moved to a different spot

Figure 5 shows an example of the self-healing. The black eCells are eCells which have been assigned a function. The grey eCells are spare-eCells. The numbers in each of the eCells indicates their unique identifier, which refers to the gene they are implementing. In this case eCell with identifier 5 is faulty. Since all eCells contain the eDNA, all that is needed to move the functionality to a spare-eCell is to change the mapping of identifier 5 to a spare eCell. The eCells will then autonomously reconfigure themselves to communicate with the spare instead of the original and the application can continue execution unabated. Observe that due to the moving of functionality the communication paths between eCells might be longer. Consequently, the only side-effect of the self-healing is possibly slightly longer communication time.

The final version of the eDNA architecture is aimed towards an ASIC implementation to provide a customized high speed eCell processor design. The prototype used in this study is solely used to study the potential performance of the final solution as well as learn about improvements necessary to be implemented in the final ASIC solution.

Further information about the eDNA architecture can be found in [1,2,14].

4. Fourier Transform Spectrometer (FTS) Application

Contrary to other spectrometers an FTS can analyze a gas using a broadband light source in one scan. This has the benefit that the instrument doesn't necessarily have to carry a light source, but could use the sun as light source. This of course will save power, instrument area and weight. But even if a special light source were required it would still save power, because the light source doesn't have to be tuned to emit different wavelengths. Also the light source would need to be on for a shorter period of time due to the parallelism inherent in the FTS.

The standard way of building an FTS is to use a Michelson interferometer to create the interferogram. However, a Michelson interferometer is using movable mirrors to vary the optical path difference. Consequently, using an FTS in space applications based on a Michelson interferometer design is considered a risk. The reason is that the immense g-forces applied during launch might misalign the mirrors, which might cause a complete loss of function or uncertainties in the precision of the returned interferogram.

With that motivation, NASA's Jet Propulsion Laboratory and Vescent Photonics Inc., built an FTS where the Michelson interferometer is replaced by a highly tunable Liquid Crystal Waveguide (LCW) [10,11]. The schematic of this FTS instrument is depicted in Figure 6.

It works like this: (1) incoming light is polarized at a 45 degree angle, causing it to split into TE and TM modes (Figure 6 left), (2) the light then enters the LCW, where a voltage is applied to an electrode, known as the OPD electrode on the LCW. Changing this voltage causes one of the modes to travel a farther distance than the other (Figure 6 mid), (3) the light exits the LCW and hits a beam combiner (Figure 6 right). Because one of the modes now is slightly out of phase with the other mode, the light received

at the detector is at a lower intensity than the light, which entered the LCW. This is exactly the same effect as the Michelson interferometer achieves, but with no moving parts! (4) By applying an FFT to the signal read from the detector the spectrum will be produced.

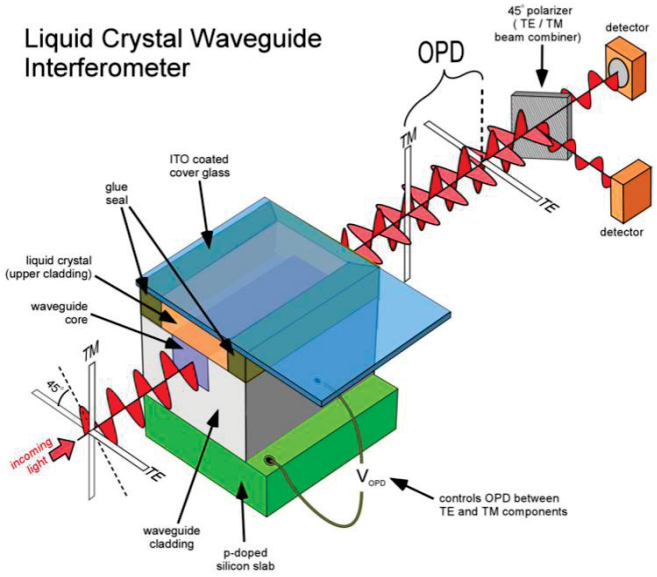

Figure 6 FTS instrument developed by NASA's Jet Propulsion Laboratory and Vescent Photonics Inc.

FTS Data Operations

A LabVIEW program is used to control the FTS and do the data processing of the detector readings.

The computational steps the program completes are:

1. Compute voltage for the OPD electrode.
2. Apply the voltage and read and process the voltage reading from the detector.
3. Process the voltage reading and do an FFT on it.

We will implement step 1 and 2 on eDNA.

5. INTEGRATION OF FTS APPLICATION IN EDNA

The first step to integrate the FTS application on eDNA is to move eDNA from its current Virtex 5 FX-130T FPGA to an embedded system known as a CompactRIO platform [12] from National Instruments. The CompactRIO platform consists of a PowerPC at 800 MHz, a Xilinx Virtex 5 LX-110 FPGA, an analog input module and an analog output module. The basic architecture of a CompactRIO platform is seen in Figure 7. A host PC programs the PowerPC and FPGA using LabVIEW. The FPGA can be used to run dedicated applications particular suited for FPGA implementation as well as be used to interact directly with the I/O modules. The I/O modules can be customized depending on the application. In our case the I/O modules are analog I/O modules to supply the electrode voltage and read the recorded voltage signal from the detector.

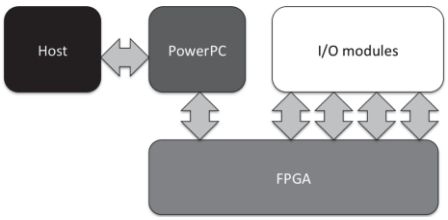

Figure 7 CompactRIO architecture

The eDNA architecture will be integrated on the Virtex 5 FPGA of the CompactRIO system. The eDNA architecture exists as VHDL code, which can be integrated into LabVIEW by using IP integration tool known as Component Level IP Node (CLIP node) [13], which is used to interpret the top-level VHDL file of a design.

eDNA architecture ⇔ LabVIEW interface

When using CLIP the top-level ports of the VHDL code becomes a LabVIEW port object, which you can read/write from/to just as you would use an I/O module. But the eDNA architecture has a special handshaking protocol that needs to be observed in order to communicate with it. Figure 8 shows a schematic of the eDNA architecture, its environment and the handshaking protocol.

eDNA needs to be told when data is ready to be read and also when the environment is ready to receive data. When data is ready to be input to eDNA, the environment asserts the *data_in_rdy* signal and provide the data. When the environment is ready to receive data, it asserts the *rdy_for_data* signal and checks whether eDNA output a package identifier (*idnt*) which is not equal to 0, (since no idnt are allowed to be zero). Whenever the environment detects that the idnt is not zero, it waits 8 more clock cycles and the data is ready. This is because the NoC of eDNA has a link width of 1 bit in order to save NoC area, so data has to be read out in serial. Observe, that just because the environment assert the *rdy_for_data* signal doesn't mean that eDNA will provide data.

This handshaking is put into a separate sub-VI for ease of use. Whenever one want to send data to eDNA the inclusion of this sub-VI will take care of the handshaking. Observe that execution will not be allowed to continue until the sub-VI is done, which is when eDNA has produced an output package based on the data.

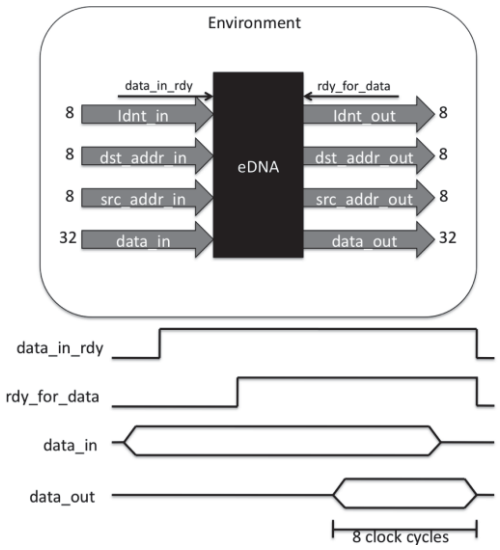

Figure 8 eDNA environment handshaking

eDNA cannot access the I/O modules directly, it has to use the LabVIEW FPGA to read/write the analog I/O. In case of a read, LabVIEW will simply send the data to eDNA using the above mentioned sub-VI. This is not 100% efficient since each time data will need to be sent to eDNA, a load package needs to be created and sent. In [2] we discovered that one package transfer from one eCell to another takes 20 clock cycles, so we're expecting a considerable performance penalty because of this.

In case of a write, LabVIEW will simply read the data from eDNA and then send it to the analog output.

Now it is also possible to download an eDNA program to eDNA by simply reading a file containing the eDNA genes and then send gene-by-gene the program to eDNA via the handshaking sub-VI.

FTS application in the eDNA architecture

The FTS application for eDNA consists of two parts:

1. Voltage ramp generation

2. Averaging of the voltage reading from the detector

The program code for these parts was provided as LabVIEW code from Vescent Photonics. In order to implement this in eDNA we had to rewrite the LabVIEW code using the eDNA programming language.

eDNA: Voltage ramp generation

The ramp is a simple linear voltage ramp going from 0 to 10 volts with a step size of 0.0025V, i.e., 4000 data points in total. The ADC of our analog output module take a 16-bit signed value and convert it to a voltage from 0-10V. Consequently, if the data were to be stored in a memory, it would only occupy 8KB of memory. However, in the current state of the eDNA architecture prototype we don't have that much memory, so eDNA will have to calculate the next ramp-step each time a new voltage ramp value is needed.

The eDNA ramp program can be seen in Figure 9.

```
i = 3999 - k
if i != 0 then
  k = k + 1
else
  k = 0
endif
Ramp = k*voltage_step
```

Figure 9 eDNA voltage ramp program

We had to make a small adaptation due to the early nature of the prototype, which is that currently eDNA can only compare to 0. This is why we use `i = 3999 - k`. This program uses 5 eCells.

When the program is run 4000 times, all ramp values will have been generated and the ramping is done.

eDNA: Averaging

The original program by Vescent Photonics, samples 160,000 points, which is downsampled to 4,000 points, i.e each point is averaged 40 times. However, to ease the hardware implementation we chose to implement the division as a right-shift, therefore the closest we can do is average each point 32 times. The eDNA program for the averaging can be seen in Figure 10. Another adaptation had to be made due to the state of the eDNA prototype, which is that if-then-else only support statements in both the true and false case, therefore we had to implement a dummy case for the true part. This implementation also uses 5 eCells.

```
sum = sum + detector
i = 31 - i
if indexdata != 0 then
  zero = zero + 1
else
  avg = sum >> 5
endif
```

Figure 10 eDNA averaging program

This program is run 32 times for each averaging. Each iteration eDNA is supplied a new *i* value and a new *detector* value from the analog input module. Furthermore, the sum is also reset before each start of the 32 iterations. This is done by LabVIEW sending a load package to eDNA with the sum reset to 0.

The two programs seem very similar, but in practice they are not. The eDNA ramp program is solely a source type of program – where the environment doesn't provide any input to it apart from activating it. However, the eDNA averaging program is acting as a sink and a source, since the environment will have to stream the detector reading to the eDNA architecture. This demonstrates that eDNA supports two different ways of interacting with its environment.

Finalizing the eDNA programs for deployment

Finally, the programs are transfered to our eDNA Software Toolkit which compiles the programs into a gene-package of the form seen in Figure 11 (note the x's denote that the value is hexadecimal). This is in fact a complete package, which is ready for sending. When an eCell receives such a gene package it will only store the 8-bit gene part; the rest is used for the package transfer. Depending on the number of connections each line in the eDNA program (except `else`, `endif` and `endwhile`) in the program takes up 4 of these genes times the number of connections to other cells, so that would be 32 bits pr. connection pr. cell. Typically a cell has no more than 2 or 3 connections. The eDNA is consequently very compact but of course scales linearly with application size.

The four 8-bit gene codes used to describe one line of eDNA program code is called a *gene-set*. The first 8-bit gene code is a program counter, which tells eDNA where to read the next gene-set from. The second 8-bit gene code is a relative address to the eCell that this gene needs to target. The third 8-bit gene code is the identifier, which the eCell will put on the package for the eCell described in the second gene and finally, the fourth 8-bit gene code is an ALU opcode telling the eCell which ALU operation to perform. If the eCell needs to forward the result of the ALU operation to more than one eCell, the ALU opcode of the next geneset will contain the code "00". This tells the eCell to not execute the operation, but simply take the result from the result register.

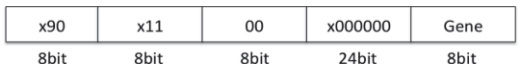

Figure 11 eDNA gene package

6. eDNA Performance Analysis

In this section we want to investigate the performance of eDNA regarding:

1. FPGA area usage
2. Execution time without faults
3. Execution time in case of faults
4. Self-healing time
5. Data maintenance

Comparison with FPGA implementation

The FPGA implementation which we compare eDNA against is a LabVIEW FPGA implementation of the same functionality in the smartest way.

This means that the ramp is implemented as a LUT containing the scaled ramp values (because the LUT can only take integer values), which are then rescaled upon reading.

The averaging is implemented as a For-loop, which adds 32 numbers and which are finally right shifted by 5.

The VIs are seen in Figure 12.

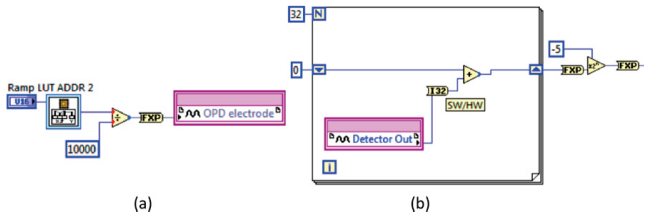

Figure 12 (a) FPGA ramp, (b) FPGA avg

FPGA area usage

Table 1 shows the FPGA area usage for the two applications.

Metric	FPGA ramp	eDNA ramp	FPGA avg	eDNA avg
# slices	5340	31773	3801	21483
# FFs	3765	17950	2774	11310
# LUTs	4659	28926	3290	18902

Table 1 FPGA area usage

For both applications eDNA uses approximately 6x more slices, 4.5x more flip-flops, and 5.5x more LUTs. This a little better than expected since the array used to implement the application is a 3x3, meaning that we have 9 eCells which all contains an ALU with a 32-bit adder and a 32-bit shifter, consequently, 8 more adders and 8 more shifters than the FPGA implementation uses.

The reason why the area of eDNA varies from application to application is because in LabVIEW more non-reentrant eDNA subVIs had to be added to take care of the streaming of the data read from the detector.

Execution time without faults

The results presented in this section are the execution time of running the ramp and averaging application without any analog I/O. The execution time for the ramp is the time it takes to calculate one ramp value. The execution time for the averaging is the time it takes to calculate the average of 32 integers.

FPGA RAMP	EDNA RAMP	FPGA AVG	EDNA AVG
41US	242US	2.42US	219US

Table 2 Execution time without faults

The FPGA ramp was implemented as a lookup table holding scaled integer values representing the voltages for the ramp. This means that we have to divide the output of the lookup table with the scaling factor. For the ramp, eDNA is 6x slower.

The FPGA averaging was implemented using an adder, which accumulated the 32 readings of the detector and finally, shifted the result 5 bits to the right. This is a very fast operation and consequently, eDNA is 90x slower. The main contributor to this time is the fact that eDNA uses 20 clock cycles pr. package transfer. The FPGA implementation can be done in 97 clock cycles. So it is clear that, since the eDNA averaging consists of 5 eCells, we will use a lot of time sending packages back and forth.

The reason why this difference is less pronounced for the ramp is due to two reasons:

1. The FPGA implementation is a memory operation and consequently is slow.

2. The eDNA program is only run once pr. ramp as opposed to 32 times for the averaging. This means that the data transfer penalty is not as visible.

Execution time in case of faults

Now we want to study the execution time in case we have faults as well as the effect of the fault.

The first fault scenario we will look at is depicted in Figure 13. Figure 14 shows the execution time for the eDNA averaging in case of an injected fault at sample nr. 1000 at the eCell shown in Figure 13. Note that the execution time displayed on the y-axis includes analog input reading plus additional logic to control the fault injection. These results should be normalized using the results from Table 2.

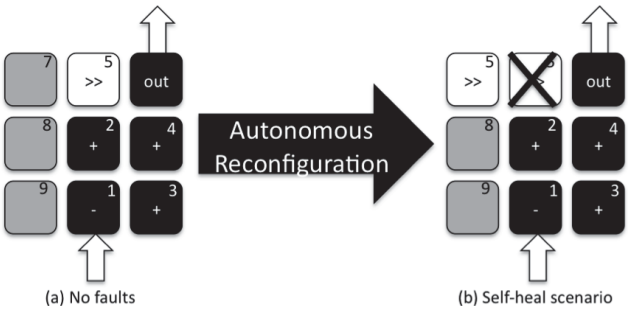

Figure 13 Fault scenario 1

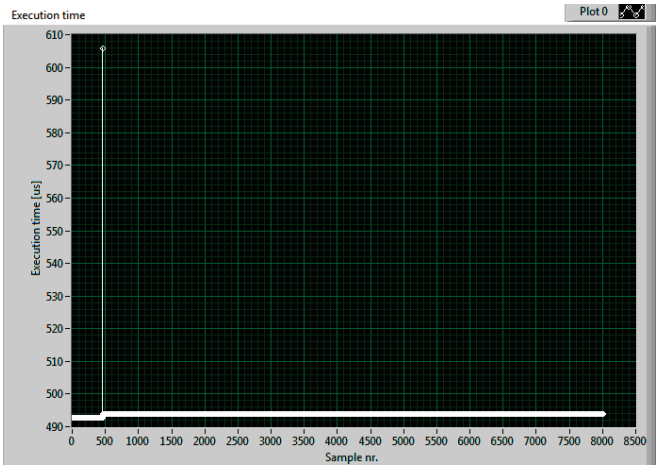

Figure 14 Execution time of eDNA averaging in case of a fault at sample nr. 1000

When the fault occurs, we see that the self-healing time is approx. 116 us higher than regularly. The exact same time is seen for the eDNA ramp application. This doesn't depend on the application but on the number of eCells, since the time reflects what is used to reconfigure. We also see that after the fault is repaired eDNA uses roughly 1 us more pr. sample. The reason for this is that eCell 5 which was faulty is now moved further away from the other eCells as seen in Figure 13.

Instead of replacing eCell 5 with eCell 7 we will now replace eCell 5 with eCell 9, which is even further away. This yields the execution time curve seen in Figure 15. After the fault the execution time is now 15us worse – just because of a slight change in chosen spare eCell. This stresses the fact that our NoC is very expensive – since all that is changed from Figure 14 to Figure 15 is the placement of one eCell.

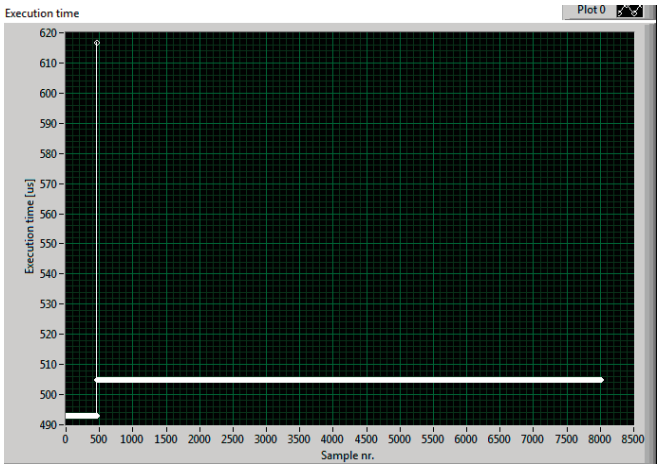

Figure 15 Execution time when picking a slightly worse eCell

As described in section 3, we have not yet implemented the fault detection algorithm. This means that we at the present prototype are incapable of always reconstructing the right data. We rely on eDNAs ability to reconstruct constants. Figure 16 shows the data output for fault scenario 1 except that the fault is introduced at 450 us. The amplitude point going to -6.5 is due to the fact that eDNA is unable to recover the right value of the *sum* when the fault occurs (Figure 10 – eCell 5 is doing: `avg = sum >> 5`). But as soon as the next data point is read the sum will be reset and eDNA will be calculating correctly again. Fortunately, eDNA notifies the user of the fault occurrence, by outputting a package with a special header, explaining where the fault occurred and where the new eCell was placed. In this way a filter could be applied, which filter this data from the original data set. This is stressed when we look at what happens for faults in the eDNA ramp. Figure 17 shows the ramp output of eDNA when a fault is introduced in an eCell at ramp nr. 450. The result is that the voltage applied drops to 0. Figure 18 shows the impact on the interferogram. Clearly, with the current state of the prototype it would be necessary to filter points which eDNA mark as faulty from the original data set.

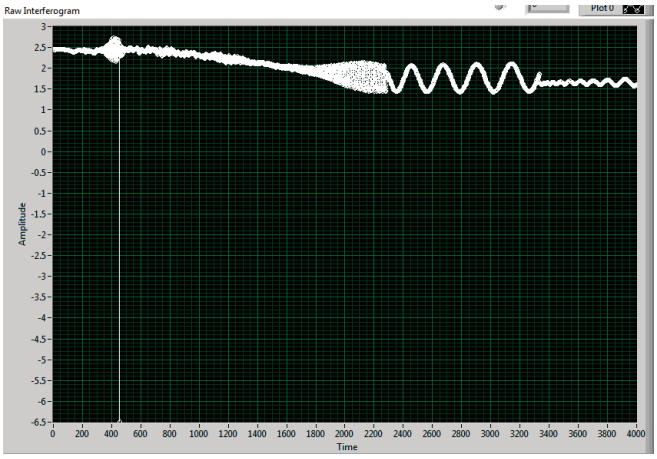

Figure 16 Impact on data when eCell 5 is faulty at sample nr. 450

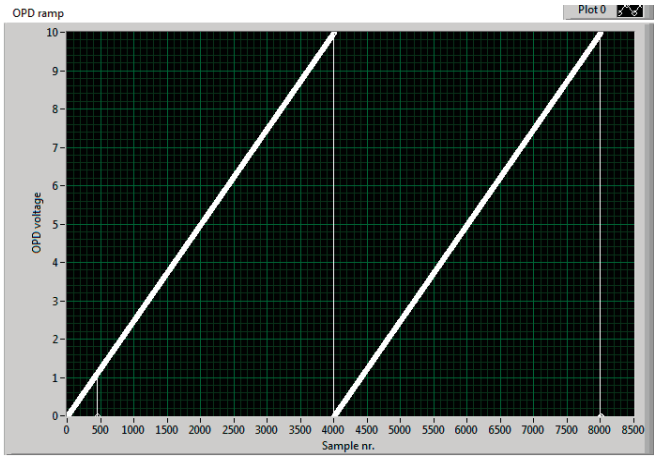

Figure 17 Fault in eDNA ramp at ramp nr. 450

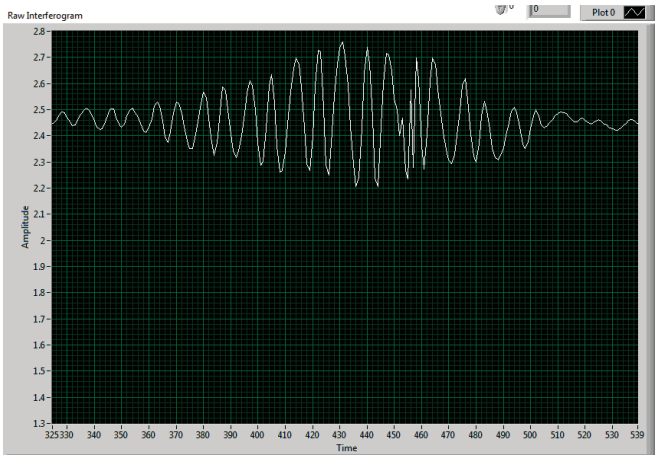

Figure 18 Impact on interferogram of fault in eDNA ramp

Result discussion

It should be noted that the comparison done in this paper is not a completely fair comparison to eDNA, because eDNA is targeted to become a new type of FPGA platform.

Consequently, eDNA is implemented as an FPGA platform on top of another FPGA platform. This would give us a unfair penalty especially for the routing, which already is a bottleneck for eDNA. However, the eDNA prototype is currently implemented on an FPGA and not an ASIC, so the results from that study is future work.

7. CONCLUSION

This work describes the first implementation of the eDNA architecture in a real world application – a Fourier Transform Spectrometer (FTS). The integration of the eDNA prototype into the CompactRIO embedded system was fast and unproblematic. We implemented the control and dataprocessing of the FTS on eDNA and learned that autonomous self-healing comes at a high cost. However, we have good reason to believe that we can decrease this cost a lot for several reasons: (1) We have a huge (20 clock cycles pr. package transfer) communication overhead in our Network-on-Chip infrastructure. We hope to improve this by making wider links, consequently eliminating our current sequential bit-by-bit package transfer, (2) The prototype is aimed at an ASIC implementation to be a new type of FPGA. Consequently, the current prototype is an FPGA architecture on top of another FPGA architecture which is bound to give a performance penalty especially for routing which is already a punishing eDNA. (3) The PicoBlaze microprocessor is only 8-bit, consequently wasting 3 clock cycles pr. operation. We would like to replace this with a dedicated ASIP.

Another important step for eDNA is the implementation of the fault-detection algorithm in order to make eDNA truly autonomous. Furthermore, the work in this paper shows that the self-healing time is a fraction of the execution time of an application.

ACKNOWLEDGEMENT

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology during Michael Reibel Boesen's 5-month visit to the Jet Propulsion Laboratory as a part of his PhD-education at the Technical University of Denmark – Informatics department under a contract with the National Aeronautics and Space Administration. The visit was made possible with financial support from the Technical University of Denmark, Oticon Fonden and Otto Mønsteds Fond.

REFERENCES

[1] Boesen, M.R., Madsen, J.: eDNA: A bio-inspired reconfigurable hardware cell architecture supporting self-organisation and self-healing. Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware Systems (2009) 147–154.

[2] M. R. Boesen, P. Schleuniger, and J. Madsen. Feasibility study of a self-healing hardware platform. Proceedings of the 2010 Conference on Applied Reconfigurable Computing, 2010.

[3] Mange, D., Sipper, M., Stauffer, A., Tempesti, G.: Toward robust integrated circuits: The embryonics approach. Proceedings of the IEEE 88(4) (2000) 516–543

[4] Stauffer, A., Rossier, J.: Self-testable and self-repairable bio-inspired configurable circuits. 2009 NASA/ESA Conference on Adaptive Hardware Systems (2009) 155–162

[5] Plaks, T., Zhang, X., Dragffy, G., Pipe, A., Gunton, N., Zhu, Q.: A reconfigurable self-healing embryonic cell architecture. International Conference on Engineering of Reconfigurable Systems and Algorithms - ERSA'03 (2003) 134–40

[6] Samie, M., Dragffy, G., Popescu, A., Pipe, T., Melhuish, C.: Prokaryotic bio-inspired model for embryonics. 2009 NASA/ESA Conference on Adaptive Hard- ware Systems (2009) 163–170

[7] Abramovici, M., Strond, C., Hamilton, C., Wijesuriya, S., Verma, V.: Using roving STARs for on-line testing and diagnosis of FPGAs in fault-tolerant applications. Proceedings of IEEE Computer Society International Test Conference (ICSM'99), 973-982, 1999

[8] Xilinx: Microblaze processor reference guide - edk 10.1i. Xilinx User Guide UG081 (v9.0) (2008)

[9] Chapman, K.: Picoblaze 8-bit embedded microcontroller for spartan-3, virtex-ii, and virtex-ii pro fpgas. Xilinx User Guide UG129 (v1.1.2) (2008)

[10] Chao, T., Lu, T., Davis, S.R., Rommel, S.D., Farca, G., Luey, B., Martin, A., Anderson, M.H.: Compact Liquid Crystal Waveguide Based Fourier Transform Spectrometer for In-Situ and Remote Gas and Chemical Sensing. Proceedings of SPIE – International Society for Optical Engineering, 2009.

[11] Chao, T.: Electro-Optic Imaging Fourier Transform Spectrometer. Proceedings of IEEE Aerospace Conference, 2007.

[12] National Instruments: NI CompactRIO – Reconfigurable Control and Acquisition System, http://zone.ni.com/devzone/cda/tut/p/id/2856. Accessed October 10, 2010.

[13] National Instruments: Importing External IP into LabVIEW FPGA, http://zone.ni.com/devzone/cda/tut/p/id/7444. Accessed October 10, 2010.

[14] Boesen, M.R., Madsen, J. and Keymeulen, Didier: Autonomous Distributed Self-organizing and Self-healing Distributed Hardware Architecture – the eDNA Concept. Submitted to IEEE Aerospace Conference, 2011

BIOGRAPHY

Michael Reibel Boesen *is a PhD-student from the Technical University of Denmark (DTU). He earned his Master of Science in Engineering from DTU in 2008 and expects to get his PhD degree in the summer of 2011. He is the co-inventor on the patent-application for the eDNA architecture. His research interests include adaptive and autonomous embedded systems. Michael is the vice-chair of the IEEE Student Branch DTU*

Didier Keymeulen *received the BSEE, MSEE and Ph.D. in Electrical Engineering and Computer Science from the Free University of Brussels, Belgium in 1994. In 1996 he joined the computer science division of the Japanese National Electrotechnical Laboratory as senior researcher. Currently he is principal member of the technical staff of JPL in the Bio-Inspired Technologies Group. At JPL, he is responsible for DoD and NASA applications on evolvable hardware for adaptive computing that leads to the development of fault-tolerant electronics and autonomous and adaptive sensor technology. He participated also as test electronics lead, to Tunable Laser Spectrometer instrument on Mars Science Laboratory. He served as the chair, co-chair, and program-chair of the NASA/ESA Conference on Adaptive Hardware. Didier is a member of the IEEE.*

Jan Madsen *Jan Madsen is Professor in computer-based systems at DTU Informatics at the Technical University of Denmark. He is Deputy Head of DTU Informatics and Head of the Section on Embedded Systems Engineering. He is the leader of the Hardware Platforms and Multiprocessor System-on-Chip Cluster within the European Union Network of Excellence on Embedded Systems, ArtistDesign. Jan Madsen is the lead delegate for Denmark in the Governing Board of the ARTEMIS Joint Undertaking, a new pan-European research initiative for public-private partnership in Embedded Systems. He has been Program Chair for DATE (International conference on Design, Automation and Test in Europe) and Program and General Chair for CODES (International conference on Hardware/Software Codesign). Jan is the other co-inventor on the patent-application for the eDNA architecture.*

Thomas Lu *is a Senior Researcher at NASA Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA. Prior to joining JPL, he was a Senior Scientist and Chief Technology Officer in several small high-tech companies for the development of high-speed 3D imaging systems and parallel optical processors. He led the design and implementation of several generations of 3D imaging products and multi-stage automatic target recognition systems. He has co-authored two book chapters, over 50 professional papers, 6 U.S. patents and several international patents. Dr. Lu's research interest includes 3D imaging, modeling, computer vision, digital/optical image processing, pattern recognition, neural networks and hyperspectral imaging.*

Tien-Hsin Chao *is a Principal MTS and group lead of the advanced optical processing group at NASA Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California. Dr. Chao is leading a team at JPL to develop optical processing and neural network based pattern recognition system, suitable for both digital and ultra high-speed optical hardware implementation. He is also developing electro-optic imaging spectrometer technologies for NASA space exploration and defense surveillance applications. He has published more than 100 technical papers, 14 U.S. patents and co-authored 3 book chapters. Dr. Chao is a fellow of SPIE. He has co-chaired the Annual Optical Pattern Recognition Conference of the SPIE Defense and Security Symposium since 1990.*